What is Rust?

Based on Oleg Ianchenko's slides and Matthew Giordano's CSE493L

The US government wants developers to stop using C and C++

Does anyone want to tell Linus Torvalds? No? I didn't think so

The Internet Has a Huge C/C++ Problem and Developers Don't Want to Deal With It

By Alex Gaynor November 15, 2018, 8:39am

Microsoft engineer clarifies speculation around plans to 'eliminate' C, C++ languages by 2030

What is C/C++?

C

- General purpose language developed in 1972 by Dennis Ritchie (also wrote Unix)
- Known for efficiency, portability, and low-level memory access
  - Essential usage in OS and embedded systems development
- Foundation for many modern languages, including C++, Java, and Python

C++

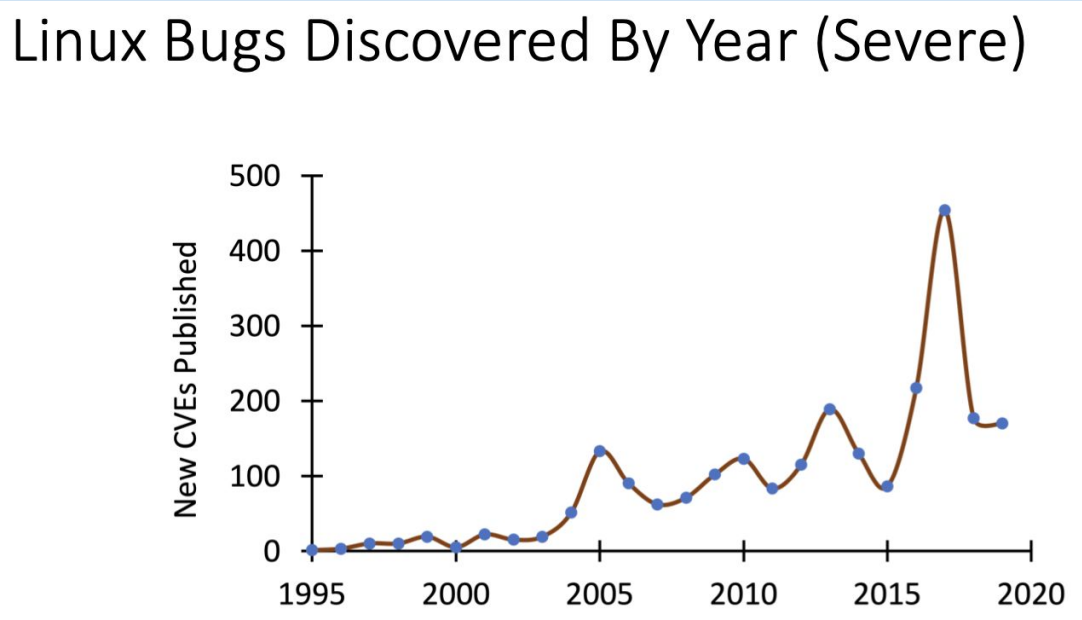
- At a high level, C but with OOP features

C is Good

- Direct program control over underlying data layout (unlike Java)
- Direct mapping of high level constructs to underlying machine instructions
  - Predictable performance
- Explicit memory allocation/deallocation
- Linux: 40 million lines of C
- Android: 12 million lines of C

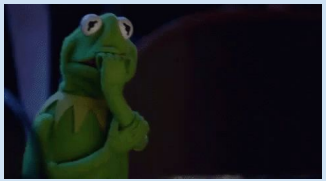
C is Bad

- Direct program control over underlying data layout (unlike Java)
- There are no “high level” constructs
- Explicit memory allocation/deallocation



C is Bad

- What if we forget to initialize data?
  - Undefined behavior
  - Will be set to whatever was in the heap/stack before!
- What if we return a pointer to data on the stack?
  - Undefined behavior
- What if we use a pointer after we free it?
  - Undefined behavior
- What if we free a pointer twice?
  - ahhhhhh



C is ***Old***

- No pattern matching
- No closures
- No first class functions
  - No currying, at all!

What about C++?

- C++ is also bad

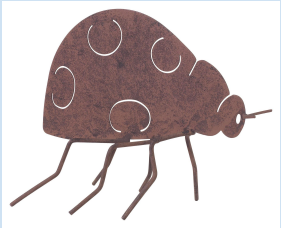
C++

```cpp
#include <vector>
#include <iostream>
int main() {
    std::vector<int> v{5, 10,};
    int &x = v[1];
    v.push_back(20);
    std::cout << x << '\n';   // segfault, or random number
    std::cout << &v[1] << ' ' << &x << '\n';
}
```

Concurrency is Exceptionally Difficult…

Concurrency bugs…

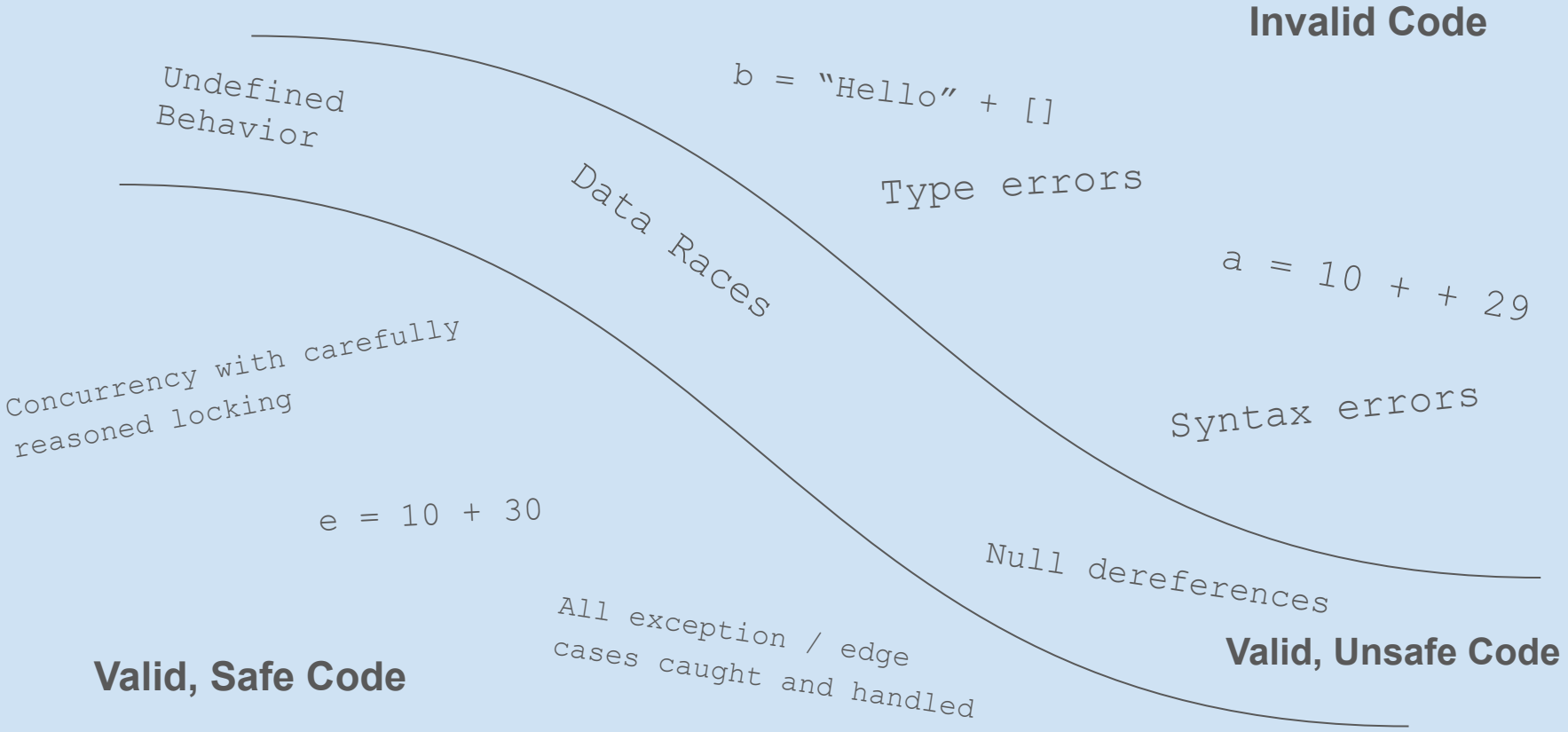
- are often runtime errors, and require extensive test suites to catch
- are hard to avoid, with concurrency being important for performance
- C does nothing to protect you against data races



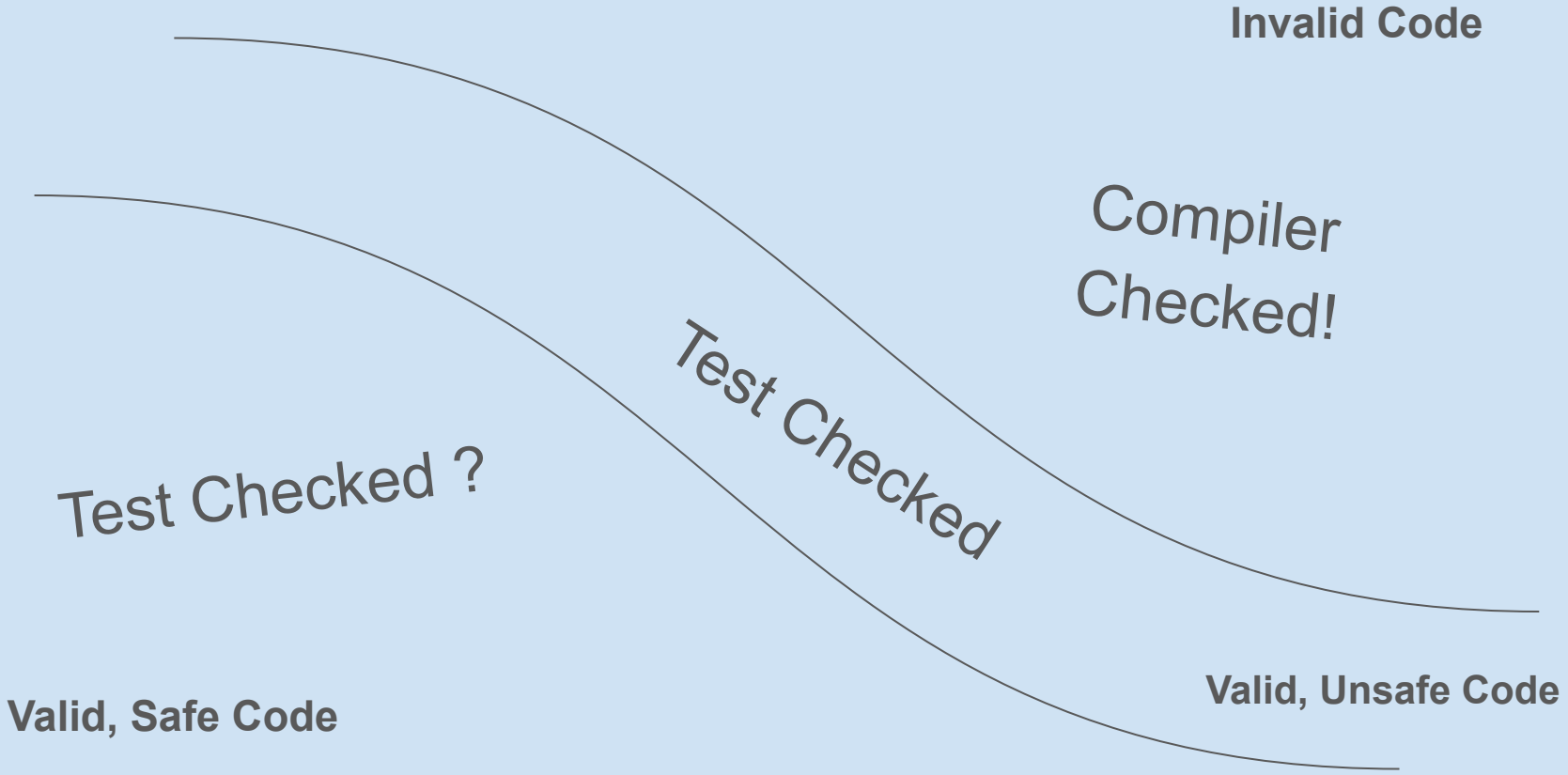
Testing provides weak guarantees for concurrent systems*, can we do better? How?

* Do not interpret this as “you shouldn’t write tests for concurrent programs”!!!!! Always test your code!!!

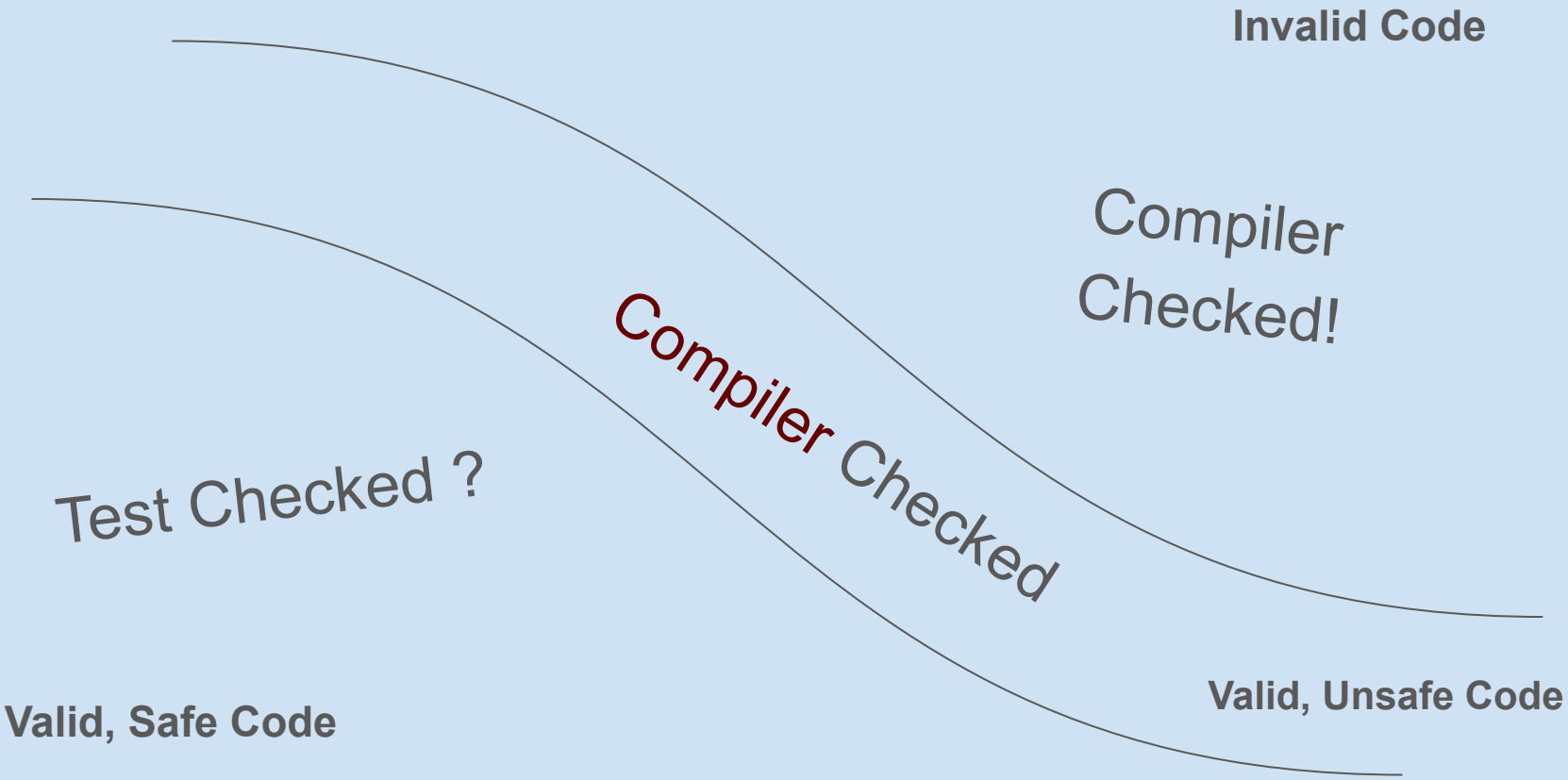
The Categories of Code



The Categories of Code



The Categories of Code



A Stronger System

If we impose more rules onto what code is and isn't "valid", the compiler becomes our proof of correctness tool

What rules do we pick? What problems do we want to solve?

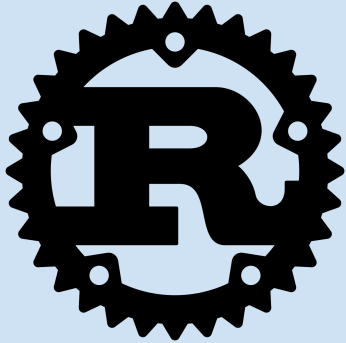
Introducing Rust!

Rust is a modern systems programming language focusing on **safety**, **speed**, and **concurrency**. It accomplishes these goals by being memory safe without using garbage collection.

– Rust By Example

Rust programmers are called ‘Rustaceans’

Rust



- Created in 2006 by Graydon Hoare (no relation)
  - Sponsored by Mozilla in 2009
  - Multi-paradigm, general purpose programming language
  - Adopted by major companies and governance via Rust Foundation
  - **Rust became the second ‘main’ language in Linux Kernel 6.1**

- Characteristics
  - Aims to support efficient, fearless, concurrent systems programming
  - Memory safe with rich type system
  - **Ergonomic developer experience**
  - Interoperable with C/C++

Data Race

2 instructions from different threads access the same memory

at least one is a write

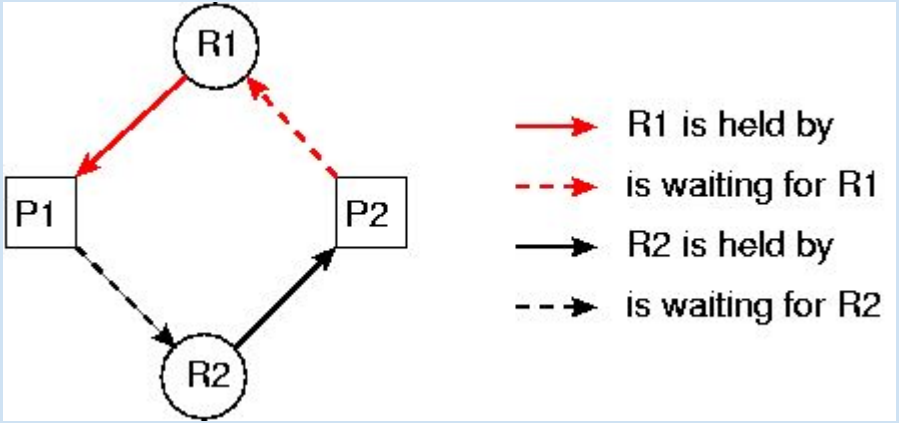
no synchronization mandating an order between the accesses

Deadlock

How to synchronize accesses so there is an order?

- Locking!

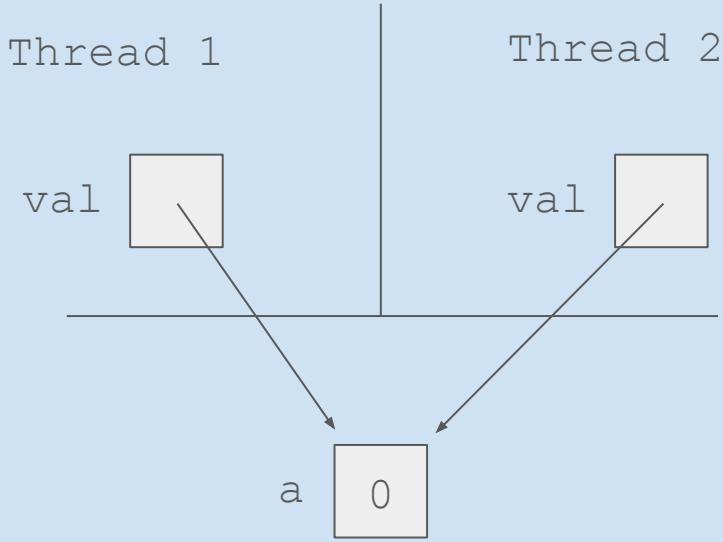
Locking isn’t always safe though.



A Stronger Concurrent System

Concurrency problems arise when data is accessed and modified at the same time.

```
fn incr(&val):
    val += 1


fn main():
    a = 0
    thread_run(incr, &a)
    thread_run(incr, &a)
    assert a == 2
```

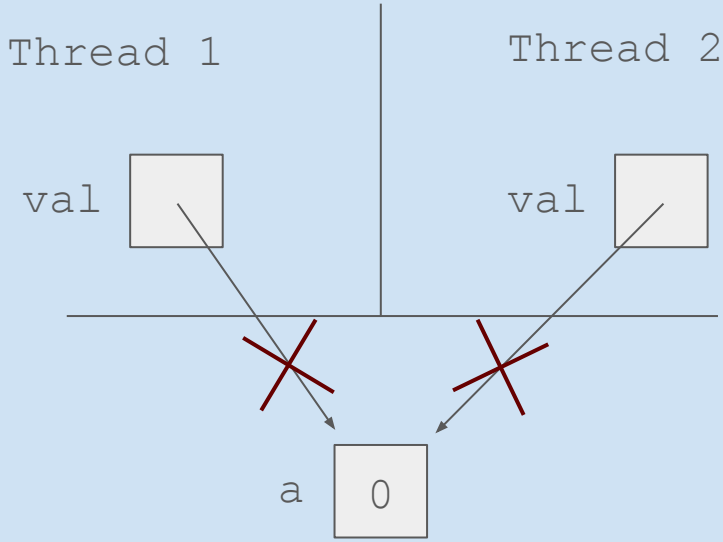


A Stronger Concurrent System

Concurrency problems arise when data is accessed and modified at the same time.

```
fn incr(&val):
    val += 1


fn main():
    a = 0
    thread_run(incr, &a)
    thread_run(incr, &a)
    assert a == 2
```

Compiler can detect this!



Disallow creating multiple ways of mutating a single piece of data!

A Stronger Concurrent System

- Multiple access is only a problem if you are planning to **write** data via a reference. Let's introduce "mutable references" (`&mut`) and "immutable references" (`&`).

```
fn incr(&mut val):
    val += 1


fn main():
    a = 0
    thread_run(incr, &mut a)
    thread_run(incr, &mut a)
    assert a == 2
```

Note, we now need to tell the compiler to expect a reference it can write to

Two mutable references → compiler error!

A Stronger Concurrent System

- Multiple access is only a problem if you are planning to **write** data via a reference. Let's introduce "mutable references" (`&mut`) and "immutable references" (`&`).

```
fn func(&val):
    print(val)


fn main():
    a = 0
    thread_run(func, &a)
    thread_run(func, &a)
    assert a == 2
```

Compiler knows we won't be writing to `&val`

Two "reader" references → this code is race safe → no compiler error!

A Stronger Concurrent System

- Multiple access is only a problem if you are planning to **write** data via a reference. Let's introduce "mutable references" (`&mut`) and "immutable references" (`&`).

```
fn incr(&mut val):
    return val + 1


fn main():
    a = 0
    thread_run(incr, &a)
    thread_run(incr, &a)
    assert a == 2
```

"Read only" references, but you are trying to write to a variable!

This is a type error!

A Stronger Concurrent System

- Multiple access is only a problem if you are planning to **write** data via a reference. Let's introduce "mutable references" (**`&mut`**) and "immutable references" (**`&`**).

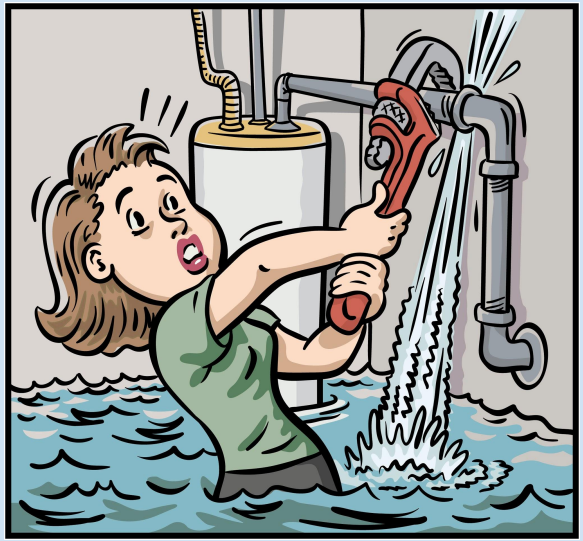
```
fn func(&val):
    print(val)


fn main():
    a = 0
    thread_run(incr, &mut a)
    thread_run(incr, &mut a)
    assert a == 2
```

You can use a "write" reference as just a "read" reference, the compiler can coerce a `&mut` down to just a `&`

A Stronger Memory System

Can the compiler help us manage memory?

- Java, Python, Go, C# are all garbage collected (GC) languages, GCs are slow.
- C / C++ require explicit freeing of memory, which is hard to do correctly at times.



A Stronger Memory System

Memory is safe to access when assigned to, and freed when we are done using it.

- Can we reason about when we can use certain memory?

```
fn main() {
    int* a = (int*) malloc(sizeof(int));
    print(*a);
    *a = 10;
    /* a bunch of operations */
    print(*a);
    free(a);
    print(*a);
}
```

Is this a safe use of `a`?

A Stronger Memory System

Memory is safe to access when assigned to, and freed when we are done using it.

- Can we reason about when we can use certain memory?

```
fn main() {
    int* a = (int*) malloc(sizeof(int));
    print(*a);
    *a = 10;
    /* a bunch of operations */
    print(*a);
    free(a);
    print(*a);
}
```

Is this a safe use of `a`?

No, it is yet to be initialized.

A Stronger Memory System

Memory is safe to access when assigned to, and freed when we are done using it.

- Can we reason about when we can use certain memory?

```
fn main() {
    int* a = (int*) malloc(sizeof(int));
    print(*a);
    *a = 10;
    /* a bunch of operations */
    print(*a);
    free(a);
    print(*a);
}
```

Is this a safe use of `a`?

A Stronger Memory System

Memory is safe to access when assigned to, and freed when we are done using it.

- Can we reason about when we can use certain memory?

```
fn main() {
    int* a = (int*) malloc(sizeof(int));
    print(*a);
    *a = 10;
    /* a bunch of operations */
    print(*a);
    free(a);
    print(*a);
}
```

Is this a safe use of `a`?

Yes!

A Stronger Memory System

Memory is safe to access when assigned to, and freed when we are done using it.

- Can we reason about when we can use certain memory?

```
fn main() {
    int* a = (int*) malloc(sizeof(int));
    print(*a);
    *a = 10;
    /* a bunch of operations */
    print(*a);
    free(a);
    print(*a);
}
```

Is this a safe use of `a`?

A Stronger Memory System

Memory is safe to access when assigned to, and freed when we are done using it.

- Can we reason about when we can use certain memory?

```
fn main() {
    int* a = (int*) malloc(sizeof(int));
    print(*a);
    *a = 10;
    /* a bunch of operations */
    print(*a);
    free(a);
    print(*a);
}
```

No!

Is this a safe use of `a`?

A Stronger Memory System

Memory is safe to access when assigned to, and freed when we are done using it.

- Can we reason about when we can use certain memory?

```
fn main() {
    int* a = (int*) malloc(sizeof(int));
    print(*a);
    *a = 10;
    /* a bunch of operations */
    print(*a);
    free(a);
    print(*a);
}
```

Idea: let's associate memory with a single *owner* variable, the owner is the instance that is responsible for freeing.

`a` owns the memory that stores 10.

A Stronger Memory System

Memory is safe to access when assigned to, and freed when we are done using it.

- Can we reason about when we can use certain memory?

```
fn main() {
    int* a = (int*) malloc(sizeof(int));
    print(*a);
    *a = 10;
    /* a bunch of operations */
    print(*a);
    free(a);
    print(*a);
}
```

Observe: When **a** leaves the scope, then there is no way of accessing it's associated memory, and therefore we can clean it up!

We don't need to free anymore, as the memory will be freed automatically at the end of main!

Looks Familiar?

- Remember scope based accessibility in Ocaml?
- From a coding perspective, Ocaml and Rust have a similar scope system
- Though architecturally different under the hood
- Ocaml uses a GC to achieve the same effect as Rust

A Stronger Type System

Types are how programmers communicate with with the compiler.

- Binaries don’t have any type information in them, the processor just accesses memory via an address and an expected data offset / length

Strong types enforce invariants.

A Stronger Type System

If you create a reference to a variable, the compiler must be able to prove that the reference goes out of scope before the thing it points to does.

No casting! No undefined behavior!

Hello World in Rust

```rust
fn main() {
    println!("Hello, World!");
}
```

```rust
fn main() {
    let unit = "CSE";
    let course_num: u16 = 334;
    let term = String::from("Winter 2026");

    println!("Hello {} {}, {} edition", unit, course_num, term);
}
```

hello_cse334.rs

```
$ rustc hello_cse334.rs
$ ./hello_cse334
Hello CSE 334, Winter 2026 edition
```

Pattern Matching

```
match io::stdin().read_line() {

    Ok(line) => {
        ...
    },
    Err(reason) => {

        ...

    },

};
```

Closures

```
let mut s = String::from("hi");

let mut closure = || {
    s.push('!');
    println!("{s}");
};
closure();
println!("{s}");
```

Variables borrowed by reference

```
let mut s = String::from("hi");

let mut closure = move || {
    s.push('!');
    println!("{s}");
};
closure();
println!("{s}"); ❌ error
          ~~~
```

Move keyword forces closure to take ownership of captured variables

Why you should care

The world is leaving C/C++ in the dust

- Rust, Go, many others….

First Class Functions